Popular Computing

73

The world's only magazine devoted to the art of computing.

April 1979 Volume 7 Number 4

Annual Forecast For The Coming Decade In Computing

ed instructions falls
ion per dollar
)

Texas Instruments finally enters the personal computer field (June 1979)

Number of personal (home) computers in the U.S. exceeds 300,000 (January 1980)

Number of makers of large main frames drops to seven (November 1979)

monthly
in the
ng
to 10
979)

General Electric re-enters the computer field (their 3rd try) (October 1985)

Computing courses are required in more than half of all U.S. high schools (April 1982)

Users of microcomputers rediscover 3-address logic (February 1980)

al

More than half of all U.S. computing power is now residing in small dedicated machines (March 1981)

A new largest prime number is found (July 1980)

Hard copy mail falls below 50% of 1977 level (February 1986)

Price of a kilobyte of RAM falls below one dollar (April 1983)

speeds of supercomputers
ch one nanosecond complete
ating add time (July 1985)

Number of persons employed full time as "programmers" starts to drop (June 1981)

The world's chess champion is a computer program, just 30 years after the first 10-year prediction of it (October 1987)

The 6th prime repunit number is discovered by an undergraduate student (December 1981)

Japanese personal machines enter the U.S. market (March 1981); they feature self-servicing.

The last traditional book publisher gives up (June 1985)

PL/I is declared officially dead (December 1979)

Giant embezzling scheme-- intimately tied to computers-- involves government payments (March 1980)

PC73--2

Annual Forecast for the Coming Decade in Computing

Our covers contain the 4th of our annual sets of predictions. How have we done?

It has been observed that short-range predictions tend to be optimistic; that is, the time passes quickly and nothing happens that was supposed to happen. Long-range predictions tend to be pessimistic; the predicted event, if it comes, comes much sooner. The cross-over between these two is about six years. It is too early to tell which was we tend in our predictions.

In 1976 we predicted that there would be 250,000 personal computers by early 1980; we are now modifying that prediction. Some of our predictions have already come true: PL/I is dying rapidly; we are probably now in the 5th generation of machines; a new largest prime was discovered; automobile computers are being installed. Dome

Some have been dead wrong: no large mainframe maker has dropped out recently; the pocket computer has not appeared (and probably won't). And some (like the one a year ago regarding TI's entry into the personal computing field) were just slightly off in their timing.

It is too early to try to keep a boxscore; perhaps a tally of successes and failures will be feasible after ten years. Any fool can make accurate predictions simply by making a great many of them; the success rate must include the booboos. Then, too, there is an analogy to the work of the weather bureau; when they predict rain, does one drop constitute success for them, or must there be some bridges washed out? When the time comes, we will persuade some outsider to rate us objectively. Meanwhile, the fun continues...join in if you care to.

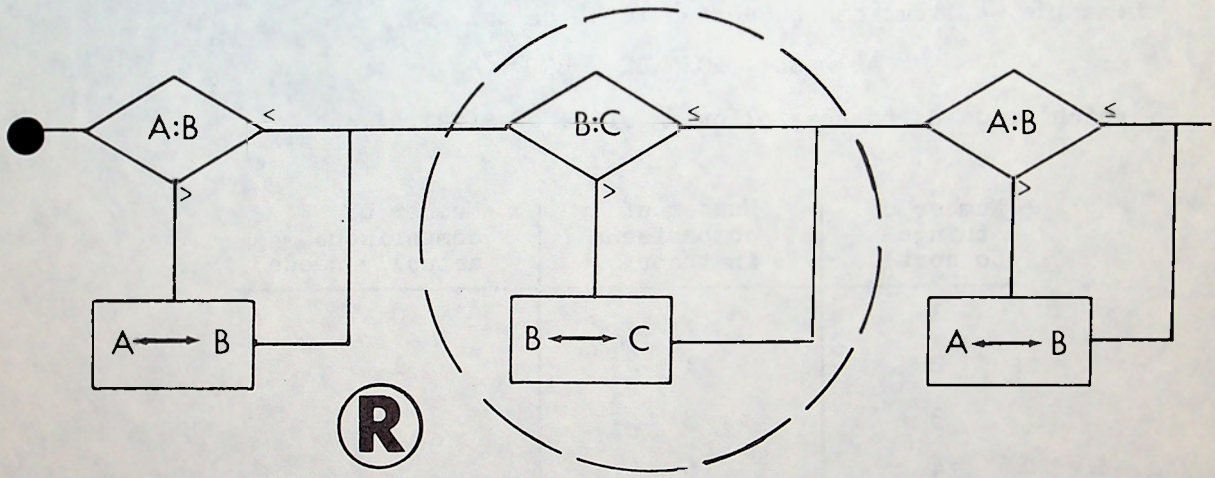
Publisher: **Audrey Gruenberger**
Editor: **Fred Gruenberger**
Associate Editors: **David Babcock**
Irwin Greenwald
Patrick Hall
Contributing Editors: **Richard Andree**
William C. McGee
Thomas R. Parkin
Edward Ryan
Art Director: **John G. Scott**
Business Manager: **Ben Moore**

POPULAR COMPUTING is published monthly at Box 272, Calabasas, California 91302. Subscription rate in the United States is $20.50 per year, or $17.50 if remittance accompanies the order. For Canada and Mexico, add $1.50 per year. For all other countries add $3.50 per year. Back issues $2.50 each. Copyright 1979 by POPULAR COMPUTING.

Eight Is Not Enough

by David Babcock

Flowchart R shows the necessary and sufficient logic to put 3 things (A, B, and C) into ascending order by direct internal sorting. The 3 comparisons and interchanges shown are not a unique set; the same result could be obtained from the comparisons of A to C, B to C, and A to B, in that order.



For brevity, let us symbolize all the logic shown in the circle of Figure R as BC. With that notation, the basic scheme for a 3-item sort is then AB BC AB.

Unlike other sorting methods where the sequence of comparisons is directed by the outcome of previous comparisons, direct internal sorting is based on a homogeneous sequence of comparisons. That is, a predefined set of comparisons is "blindly" applied without regard to any previous interchanges. Such a constrained method of sorting is important because of the ease with which it can be implemented in hardware.

3: AB BC AB

4: AB BC CD AB BC AB

5: AB BC CD DE AB BC CD AB BC AB

6: AB BC CD DE EF AB BC CD DE AB BC ...

... CD AB BC AB

(i.e., case N is formed from case N-1 by preceding the pattern by all possible comparisons from left to right.)

The basic scheme shown in the flowchart for a 3-item sort extends readily to the sorting of any number of items, as shown in Figure S. However, it is observed that this simple scheme is not the most efficient possible. For example, 4 items can be sorted with the scheme:

AB CD AC BD BC

which leads us to the following state of affairs:

Number of things to sort	Number of comparisons in theory	Number of comparisons actually needed
1	0	0
2	1	1
3	3	3
4	6	5
5	10	?

Thus, attention is drawn to the case N = 5. The problem is: What is the minimum number of comparisons and interchanges needed to sort 5 things?

(Various interesting theoretical approaches suggested that the number of comparisons needed should be 8, which might be a satisfactory solution except that then the burning question is <u>which 8?</u>)

I set out to establish the answer to the problem, using a brute force approach; that is, by trying every possible combination of 8 comparisons to sift out the combination that would work.

Note: while having all the fun of coding and running this monumental search, I discovered that Knuth had already solved the problem* and that the answer is 9.

However, there is some point to reporting on the exhaustive attack, and a side effect popped up that may be of more value than the original research.

We need some notation. We can label the positions to be sorted as follows:

A	B	C	D	E

And the ten possible comparisons can be coded this way:

0	AB	5	BD
1	AC	6	BE
2	AD	7	CD
3	AE	8	CE
4	BC	9	DE

We can form a systematic scheme for testing all combinations:

1	2	3	4	5	6	7	8

Each of these boxes represents one of the 8 comparisons to be applied to the 5 sort positions. Comparisons are to be applied in order (left to right). Each box can thus take on the values from 0 to 9, taken from the coding scheme given above. To test all possible combinations systematically, run the 8 boxes as an 8-digit decimal counter. This implies that there are 100,000,000 combinations to test, and for each such combination we must try the 120 permutations of 5 things, to insure that each of them is put into ascending order.

---

*This phenomenon--namely, discovering that Knuth has anticipated what you have just discovered--is becoming increasingly frequent, and will become even more so when the remaining four volumes of The Art of Computer Programming appear. For the result we need here, see Section 5.3.4 of Knuth.

Side note: There is no need to additionally test the 7-comparison combinations. If such a set of 7-comparisons existed, it would have been found while testing 8-comparison combinations. For example, if the first 7 comparisons had done all the work, then it wouldn't matter what the 8th comparison was, and 10 workable solutions would have been found. In all, (at least) 80 sets of level 8 solutions would have been found for each level 7 solution.

In order to decrease the amount of computer time needed to run the program, many improvements were made, both to the method of solution and to the code itself. These efficiencies can be divided roughly into four areas.

Improvement #1:

When testing a set of comparisons, not all the 120 permutations of 5 items need to be tried. As soon as a trial set of comparisons fails to order one of the permutations, no further testing needs to be done with that set. Only a "good" set of comparisons needs to be applied to all of the possible permutations (and then to only 119 of them, since the arrangement 12345 will always be in sort).

The first improvement, then, is to order the permutations in such a way that the permutation in "worst sort" is tested first; the next worst second; and so on. By doing this, most of the bad sets of comparisons will be eliminated by performing only one sort operation. This is a significant improvement to the solution (in the production run, 89% of the cases were eliminated on the first sort).

To implement this improvement requires that some measure of "out-of-sort"-ness* be applied to each of the 120 permutations. The permutations can then be ranked according to this measure. This was done and Table 1 lists the 120 permutations in the order in which they were tested in the production run.

---

*As it turns out, the development of a good measure of "out-of-sort"-ness is actually more interesting and complex than the original problem I set out to solve. More on this later.

Permutations (listed in order tested)

1	53421	31	35124	61	51324	91	25341
2	54231	32	41523	62	51243	92	52314
3	45231	33	23451	63	43215	93	51342
4	53412	34	51234	64	25314	94	32541
5	43521	35	23514	65	41352	95	52143
6	35421	36	25134	66	15432	96	32415
7	54213	37	31452	67	34215	97	24315
8	54132	38	41253	68	32514	98	42135
9	34521	39	24153	69	42153	99	41325
10	45213	40	31524	70	25143	100	14352
11	43512	41	23154	71	43125	101	13542
12	35412	42	21453	72	31542	102	15324
13	45132	43	21534	73	14532	103	15243
14	54123	44	31254	74	15423	104	23145
15	34512	45	54321	75	34125	105	31245
16	45123	46	45321	76	14523	106	13425
17	43251	47	54312	77	23415	107	14235
18	53214	48	45312	78	13452	108	12453
19	25431	49	53241	79	41235	109	12534
20	51432	50	52431	80	15234	110	52341
21	34251	51	35241	81	32154	111	42315
22	24531	52	42531	82	24135	112	15342
23	53124	53	52413	83	21543	113	32145
24	51423	54	53142	84	31425	114	14325
25	35214	55	42513	85	13524	115	12543
26	25413	56	35142	86	14253	116	21345
27	43152	57	32451	87	21435	117	13245
28	41532	58	24351	88	21354	118	12435
29	24513	59	23541	89	13254	119	12354
30	34152	60	52134	90	42351	120	(12345)

Table 1

Improvement #2:

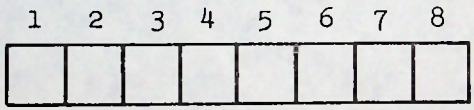
Any comparison which duplicates the work of either of its nearest neighbors is doing no useful work. In other words, there is no need to try any combination where two adjacent digits are the same.

Note that two subcases need to be considered here:

1. If the sorting can be done in 8 (and no fewer) comparisons, then the above explanation holds directly.
2. If the sorting can be done in 7 comparisons, then we have succeeded in discarding some of our good cases, but not all of them. We'll still get (at least) 73 sets of level 8 solutions for each level 7 solution which exists.

To implement this, one additional check needs to be made. When changing comparison K, if the new value equals comparison (K-1), then change comparison K again. Picture an analogy with an odometer. When a wheel is rotated one position, if its "value" is equal to the value of the wheel to its left, then rotate it again.

To compute the number of comparisons which will now be tested:



number of values which do not equal the comparison to its left.

$= 10 \cdot 9^7 = 47,829,690$ combinations, or less than half the number we started with.

Improvement #3:

Every one of the 5 number positions must be tested at least once. Any set of 8 comparisons which fails to test any one of the 5 positions can be discarded immediately without performing any sorts. If the every-position-tested test can be performed cheap enough, we may come out ahead with it. (Note that if this test is not made, all of the cases it would have rejected would have been eliminated by the first sorting case, since that case has every position out of place.)

This test was implemented in the production run, but it turned out to cost more than it gained, because so few cases were eliminated by it. The final production run of about 10 hours would have been 30 minutes shorter without this test.

Improvement #4:

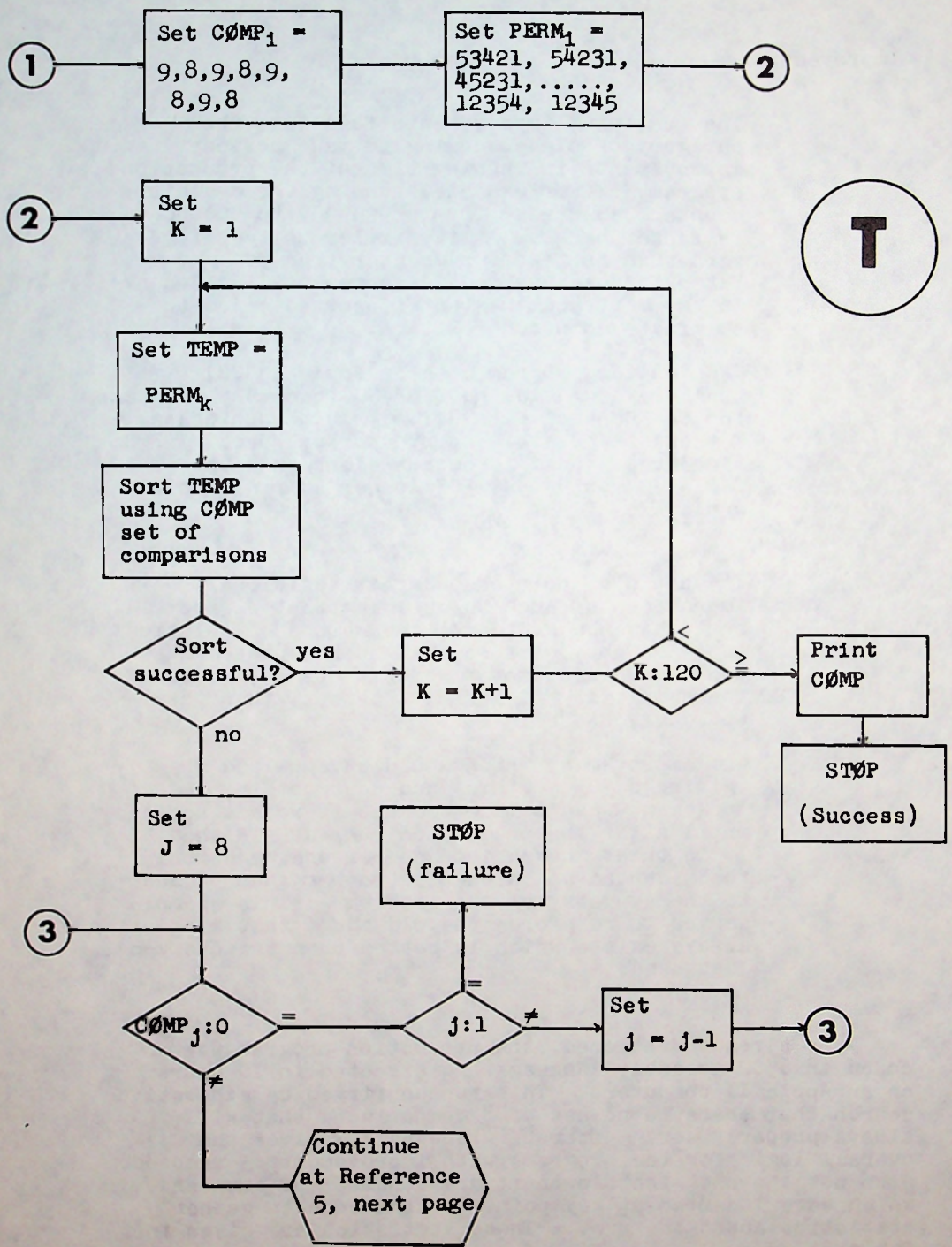
The remaining improvements fall into the category of clever coding to make obvious improvements in running time of the production program. For example, running the 8-digit counter backwards, from 999999999 to 00000000, is faster because it is simpler to test for zero than to test for greater than 9. Extensive use was made of straight-line coding in the sort and the test-if-sorted codes to eliminate loop control time.

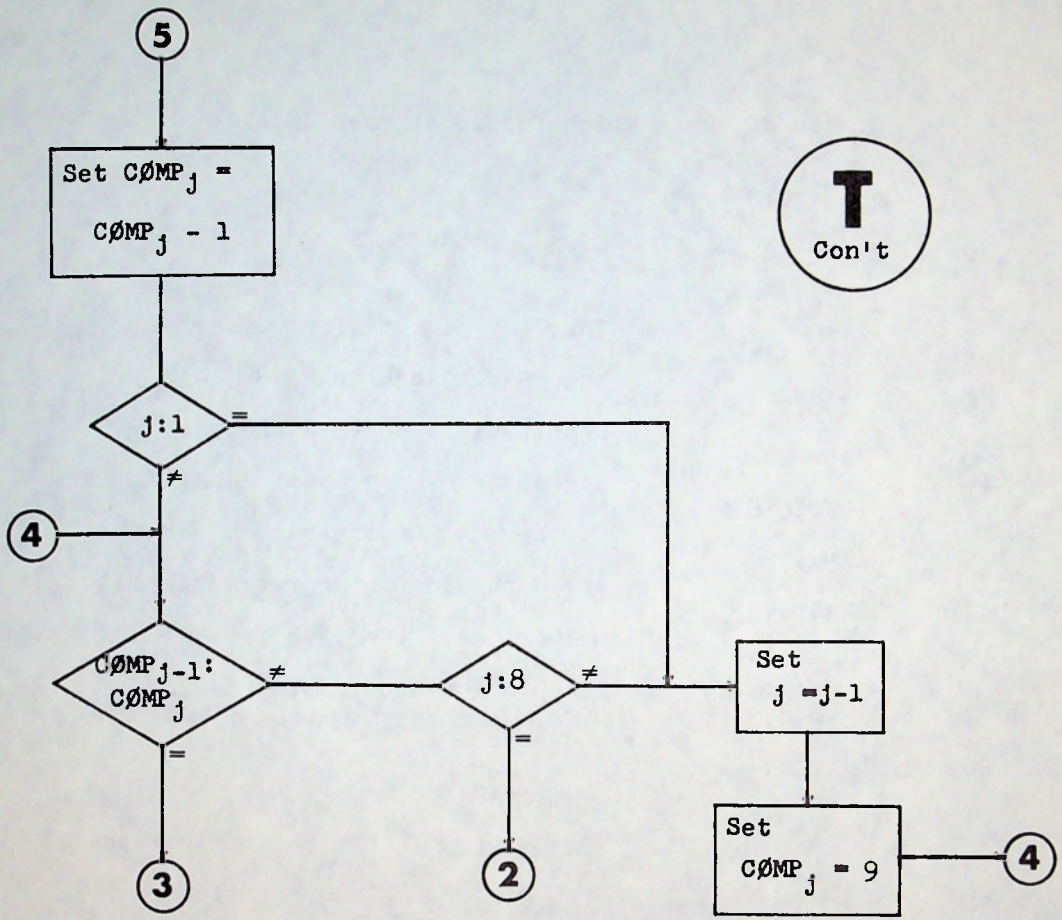
I coded the program both in Apple (6502) assembly language and COMPASS (assembly language for the Control Data 3170). In both cases, I took all possible shortcuts for speed. From timed test runs, it became clear that the CDC 3170 would take 50% longer to run than the Apple II.

It should be pointed out that additional improvements (such as refining test #2 above) could have been made to the method of solution but were not, because most such improvements would have cost more execution time (due to the complexity of making the test) than they would have saved.

One improvement which would have helped greatly (giving a 30% savings) and cost nothing to implement wasn't made because it wasn't noticed until after the production run was finished. It is this: comparison number one needs only to be tested up through 6 (rather than 9) due to the symmetry of the problem. This example serves to reinforce the old adage that a solution always exists which is better than the one you just used.

As already mentioned, the production program was coded in 6502 assembly language and executed in 10 hours on an Apple II computer. This run confirmed by exhaustive search that there is no set of 8 comparisons that will always properly sort 5 items. Flowchart T gives the overall logic for the program, with improvement #3 removed. Also not shown on the flowchart are a number of counters, which were inserted at key points in the code to gather statistics about the run. These statistics are given in Table 2.


1
Set CØMP1 = 9,8,9,8,9, 8,9,8
Set PERM1 = 53421, 54231, 45231,....., 12354, 12345
2
2
Set K = 1
T
Set TEMP = PERMK
Sort TEMP using CØMP set of comparisons
Sort successful?
yes
Set K = K+1
K:120
<
≥
Print CØMP
STØP (Success)
no
Set J = 8
STØP (failure)
3
CØMPJ:0
=
j:1
=
≠
Set j = j-1
3
≠
Continue at Reference 5, next page



In discussions of sorting, it is frequently assumed that data in strict descending order constitutes the worst case for a sorting scheme. This is sometimes true (it certainly is for bubble sorting) but not always. For example, consider the Shell sorting scheme (the logic was given in our issue 58, page 6). In outlining the method of Shell sorting, the steps were given for sorting eleven items that were in descending order. For that set of data, the sort makes 15 interchanges. But the set:

10 11 9 8 7 5 6 4 3 2 1

takes 17 interchanges, and is thus a worse case for Shell sorting.

Number of cases which failed at each level*

1	40,367,418
2	2,943,906
3	1,044,582
4	571,746
5	168,043
6	182,240
7	74,978
8	59,779
17	34,143
18	9,568
19	8,863
20	12,244
21	2,945
24	2,419
25	1,607
26	1,607
33	1,443
34	883
35	718
36	648

Table 2

That is, permutation number one (53421) caused 40,367,418 combinations to fail and they didn't have to be tested any further. Of those combinations which passed sorting permutation number one, 2,943,906 failed on permutation number two (54231) and so on.

The number of combinations eliminated for not testing all positions:

2,339,910.

Average number of levels tested:

1.23338873

*All other permutations had a count of zero.

If reverse order isn't necessarily the worst case for testing sort algorithms, then what is and how do we find it? A popular method for measuring the sortedness of a set of numbers is to count the number of "inversions." The concept is attributed to G. Cramer and is described by Knuth (in Section 5.1.1): "each inversion is a pair of elements that is 'out of sort'." For example, the set of numbers:

2 4 1 5 3

has four inversions; namely,

(2,1) (4,1) (4,3) (5,3).

Table 3 summarizes some key permutations and their number of inversions. It is clear after studying the table that using the inversion count, as defined, will not produce the optimal ordering of the permutations. The permutation with the largest number of inversions is 54321 and this is clearly not the worst case. With the center element already in its proper place, many trial sets of comparisons will "succeed" in sorting this permutation when they shouldn't. Remember that for the problem at hand, the worst case permutation is that arrangement which will eliminate the greatest number of faulty sort sets. Another approach to measuring "out-of-sort"-ness was needed.

Permutations and their Inversion Count

54321	10
53421	9
34521	8
52341	7
25431	7
35241	7
52314	6
34512	6
15432	6
31542	5
42135	4
31245	2
12435	1
12345	0

Table 3

The approach I took was to formulate my own measure which I call a displacement coefficient. To order the permutations we do the following:

1) Group the arrangements according to the number of values which are out of place. These groupings are indicated by the solid lines in Table 1. (Note that this could also have been done when using inversion counts to produce a better ordering.)

2) Within each group, rank the permutations according to the following formula:

$$dc = \sum_{i=1}^{n} (a_i - i)^2$$

(This assumes that the a values range from 1 to n)

The reasoning behind the displacement coefficient is similar to that for the variance of a set of numbers. That is, the further out of position a number is, the "harder" it is for a sort algorithm to put it back where it belongs.

From preliminary tests of the displacement coefficient, the inversion count and several other measures of sortedness, it appeared that the displacement coefficient did the most effective job of ordering the permutations for this problem. Its ordering was therefore used in the production run.

The statistics gathered from the production run (Table 2) show several interesting things:

1) The basic ordering of the permutations is very good. Permutations 5 and 6 both have displacement coefficient values of 34 so their final ordering was arbitrary. Similarly, permutations 18, 19, and 20 all have values of 28 and could have been put in a different order.

2) The gaps of permutations with zero counts seems to indicate that these arrangements are in some manner subsets of previous permutations. If permutations 9-16, 22, 23, 27-32 were discarded, almost 700,000 needless sort operations would have been saved. This strongly suggests that some type of pattern matching, applied to the 120 permutations to identify these subset cases, would be beneficial.

3) While better (at least for this problem) than the other measures of "out-of-sort"-ness, the displacement coefficient is not as good as it could be. The program was modified so that each set of sort comparisons would be tried on each of the first ten permutations regardless of previous failures. The results of this run are given in Table 4. From these statistics it appears that 45231 (not 53421) is the worst ordering of five values. The best formula for measuring sortedness should therefore place 45231 at the top of the list.

The investigation into other measures of sortedness is an area which requires more work. The research presented here has suggested an approach which needs to be pursued further. It is apparent that there is at least one category of sorting problems for which existing algorithms are not appropriate.

Number of cases which fail to sort each of the first ten permutations.

1	42,707,328
2	42,707,328
3	43,147,474
4	43,147,474
5	43,089,794
6	43,089,794
7	43,089,794
8	43,089,794
9	42,879,276
10	42,852,480

Table 4

That is, of the 45,489,780 combinations tested, 42,707,328 failed to sort permutation number one; 42,707,328 failed to sort permutation number two, etc.

This table suggests that even my initial formulation of the displacement coefficient is not optimal. Permutations number three and four seem to be the most sensitive for testing sorting schemes, and should be at the top of the list.

□□□
□□□

PC73-16

Andree's Problem

(The following problem comes from Prof. Richard Andree, the genial computing sage at the University of Oklahoma.)

It has been proved that there exist arbitrarily long strings of consecutive positive integers such that each of the consecutive integers has as a factor a perfect square greater than one (probably different squares for different integers). Putting it in more formal terms:

> Given a positive integer N, there exist strings of N consecutive positive integers each of which contains a factor which is a perfect square greater than one.

If N = 3, the consecutive numbers: 48 49 50

contain as factors: 4^2 7^2 5^2

98 99 100

7^2 3^2 10^2

For N = 4 we have, for example: 242 243 244 245

11^2 3^2 2^2 7^2

And, for N = 7:

217070	217071	217072	217073	217074	217075	217076
7^2	3^2	2^2	13^2	11^2	5^2	4^2

Your preliminary problem is to find the <u>first</u> (that is, involving the smallest integers) string of N consecutive integers each of which contains a square greater than one as a factor, for N = 2, 3,...,10.

The proof has, in fact, been generalized for higher powers than 2, so that we have:

> Given positive integers N and K, there exist strings of N consecutive positive integers each of which contains a perfect Kth power greater than one as a factor.

Your second problem is to find the smallest such strings for K = 2, 3, 4, and 5 and N = 2, 3, 4,...,10 (this is a total of 36 strings).

Here are a few more results, for checking purposes; the power factor is given below each of the consecutive integers:

3rd powers (K = 3)

N = 2:	80	81		
	8	27		
N = 3:	1375	1376	1377	
	125	8	27	
N = 4:	22624	22625	22626	22627
	8	125	27	331

4th powers (K = 4)

N = 2:	80	81	
	16	81	
N = 3:	33614	33615	33616
	2401	81	16

The search for these strings is an interesting project which will bring out the advantages of good programming practices and also the advantage of a compiler versus an interpretive system on long running problems.

Any gauche (but valid) program will smoke out the first few strings at various K levels, but it soon becomes clear that a very small amount of programming sense will cut the total time sharply--and then that even the improved algorithm may still make an unreasonable demand for computer time unless a compiler or assembler program is substituted for an interpreter (which most BASICs are).

TRY IT--you'll enjoy the challenge.

Penny Flipping Again

In issue 71 we returned to the first of the Penny Flipping problems:

> A pile of C pennies is arranged so that each penny is heads up. We define an operation FLIP(Q) on this pile, which removes the top sub-pile of Q pennies, turns the sub-pile upside down and replaces them on top of the C - Q pennies remaining. The consecutive operations:
>
> FLIP(1),FLIP(2),...,FLIP(C),FLIP(1),FLIP(2),...
>
> are repeated until the stack returns to all heads. The value of N for a given C is the number of flips required to return the stack to heads. For C = 6, for example, N = 35.

with results for the cases C = 1 through C = 240 and a conjecture that the value of N (flips) for any given value of C was one of

$$kC,\ C^2,\ C^2 - 1,\ kC - 1.$$

We hear now from another genial sage, David E. Ferguson, President of the Ferguson Tool Company, who points out:

"If n is the smallest positive integer such that

$$2^n \equiv \pm 1 \pmod{2C+1} \qquad \text{(for C coins)}$$

then if $2^n \equiv 1 \pmod{2C+1}$, the number of flips is

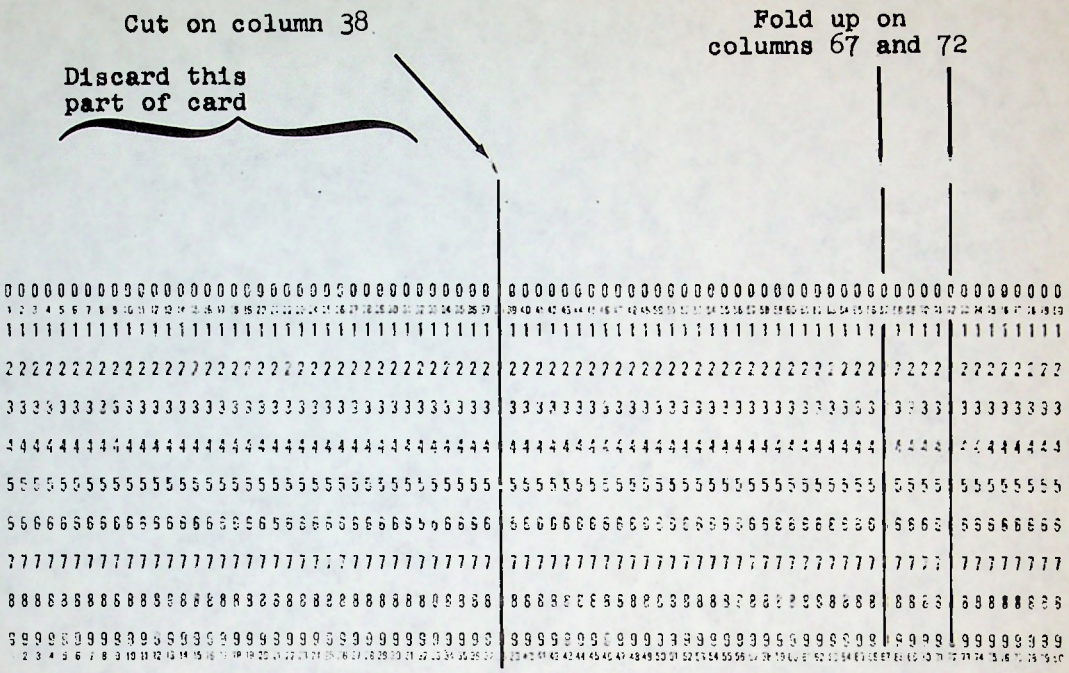
$$f(C) = nC$$

while if $2^n \equiv -1 \pmod{2C+1}$,

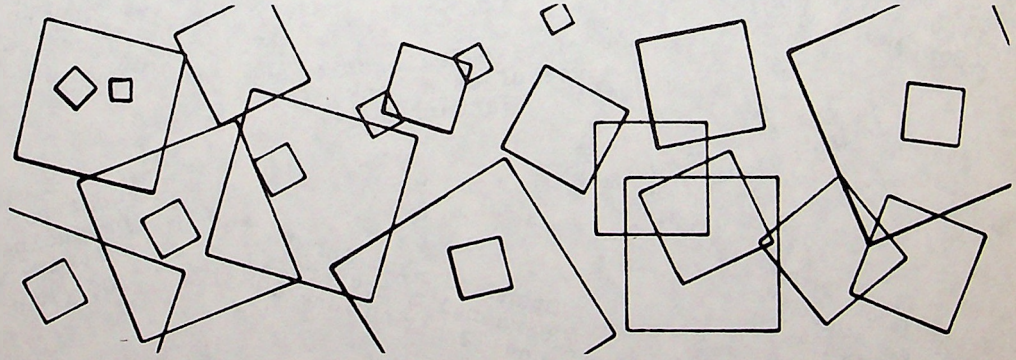
$$f(C) = nC - 1.$$

Note that n divides $\Phi(2C+1)$."

Mr. Ferguson's observation does not seem to help particularly in finding N for a given C, but it does confirm the conjecture.



Labels for cassettes in Norelco-type boxes.



PC73-20

Texas Instruments becomes No. 1 in the personal computer field (September 1980)

Cost of executed instructions falls below one billion per dollar (December 1981)

Number of personal (home) computers in the U.S. exceeds 500,000 (January 1982)

Superprogrammers now rate $50K salaries (December 1980)

The job category "programmer" has disappeared (November 1983)

Number of monthly magazines in the personal computing field drops to 8 (December 1982)

Generalized voice recognition is achieved (mid 1986)

A new "new largest prime number" is found (February 1985)

All U.S. cars have extensive on-board computer controls (September 1981)

Cost of execu
below 100 mil
(December 197

Number of makers of large main frames drops to six (September 1987)

Number of centralized large computers in the U.S. falls below 1000 (May 1987)

Number
magazine
personal
fiel
(Decembe

First microcomputer true compiler (for BASIC, of course) appears (December 1979)

The pocket calculator industry reaches 35 million units per year (September 1980)

IBM re-enters the pers
computer field with a
under-$1000 main fram
(July 1982)

Restrictive legislation on the use of data banks is enacted (January 1983)

To
re
fi

The first two companies in the personal (hobby) computing field have now given up.

Cost per line of producing complex software exceeds $100 (March 1984)

Standard word length for personal computers is now 32 bits (June 1983)

Pocket calculators with 10,000 program steps now cost $150 (April 1980)

Attention is called to another set of predictions -- some of them tongue-in-cheek -- in the January DATAMATION, page 217.